


Flare-On 3: Challenge 1 Solution

Challenge Author: Alex Rich

When running `challenge1.exe` we are presented with a password prompt, for which the program will respond to an incorrect response with “Wrong password”.



```
C:\windows\system32\cmd.exe
>challenge1.exe
Enter password:
test
Wrong password
>
```

Figure 1: Screenshot of challenge1 wrong password

Since we are looking to find a specific password, we could start off by checking strings on the challenge using the Microsoft Sysinternals “Strings” tool. Two of the most interesting strings that would be revealed are:

```
String1: x2dtJEOmyjacxDemx2eczT5cVS9fVUGvWTuZWjuexjRqy24rV29q
String2: ZYXABCDEF GHI JKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/-
```

Figure 2: Interesting strings from Strings tool output

Unfortunately, neither of these strings works as the password, so the next step would be to open this challenge up in a disassembler. The freeware version of IDA Pro will work fine in this case. The `main()` function is located at address `0x401420`.

```

; Attributes: bp-based frame
; int __cdecl main(int argc, const char **argv, const char **envp)
_main proc near

Buffer= byte ptr -94h
var_14= dword ptr -14h
var_10= dword ptr -10h
var_C= dword ptr -0Ch
hFile= dword ptr -8
NumberOfBytesWritten= dword ptr -4
argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h

push    ebp
mov     ebp, esp
sub     esp, 94h
push    0FFFFFF5h          ; nStdHandle
call   ds:GetStdHandle
mov     [ebp+hFile], eax
push    0FFFFFF6h          ; nStdHandle
call   ds:GetStdHandle
mov     [ebp+var_C], eax
mov     [ebp+var_10], offset aX2dtJE0mjJacx0emx2eczT5c0S9F00G0vTu2Vju"...
push    0                    ; lpOverlapped
lea     eax, [ebp+NumberOfBytesWritten]
push    eax                ; lpNumberOfBytesWritten
push    12h                 ; nNumberOfBytesToWrite
push    offset aEnterPassword ; "Enter password!\r\n"
mov     ecx, [ebp+hFile]
push    ecx                ; hFile
call   ds:WriteFile
push    0                    ; lpOverlapped
lea     edx, [ebp+NumberOfBytesWritten]
push    edx                ; lpNumberOfBytesRead
push    00h                 ; nNumberOfBytesToRead
lea     eax, [ebp+Buffer]
push    eax                ; lpBuffer
mov     ecx, [ebp+var_C]
push    ecx                ; hFile
call   ds:ReadFile
mov     edx, [ebp+NumberOfBytesWritten]
sub     edx, 2
push    edx
lea     eax, [ebp+Buffer]
push    eax
call   sub_401260
add     esp, 8
mov     [ebp+var_14], eax
mov     ecx, [ebp+var_10]
push    ecx                ; char *
mov     edx, [ebp+var_14]
push    edx                ; char *
call   _strcmp
add     esp, 8
test    eax, eax
jnz    short loc_40140F

; lpOverlapped
lea     eax, [ebp+NumberOfBytesWritten]
push    eax                ; lpNumberOfBytesWritten
push    00h                 ; nNumberOfBytesToWrite
push    offset aCorrect      ; "Correct!\r\n"
mov     ecx, [ebp+hFile]
push    ecx                ; hFile
call   ds:WriteFile
jmp     short loc_401406

loc_40140F:                ; lpOverlapped
push    0
lea     edx, [ebp+NumberOfBytesWritten]
push    edx                ; lpNumberOfBytesWritten
push    11h                 ; nNumberOfBytesToWrite
push    offset aWrongPassword ; "Wrong password!\r\n"
mov     eax, [ebp+hFile]
push    eax                ; hFile
call   ds:WriteFile

```

Figure 3: IDA Pro screenshot of challenge1 main()

Upon looking at the control flow of the program (shown in Figure 3), we should see two calls to

`GetStdHandle()`, an API used to retrieve the handles for the Input and Output standard I/O streams.

We can get IDA Pro to show the specific handles being requested by right clicking on the value being used as the parameter to `GetStdHandle()` (eg. `0FFFFFFF5h`) and selecting “Use standard symbolic constant”, then picking from the next dialog the constant matching `STD_*_HANDLE`. The return values from these functions, recorded in the `eax` register, are then stored in stack variables that we can rename appropriately for greater clarity.

```

push    STD_OUTPUT_HANDLE ; nStdHandle
call    ds:GetStdHandle
mov     [ebp+stdout], eax
push    STD_INPUT_HANDLE ; nStdHandle
call    ds:GetStdHandle
mov     [ebp+stdin], eax
    
```

Figure 4: Marked up calls to `GetStdHandle()` for challenge1

Next in `main()`, the first of the strings that we found originally from running the strings tool is shown getting assigned to a variable labeled by IDA as stack variable `var_10`, which we can rename to “important_string”.

```

mov     [ebp+important_string], offset aX2dtjeomyjacxd ; "x2dtJE0myjacxDemx2eczT5cUS9f0UGvWtu2Wju"...
    
```

Figure 5: Interesting string in challenge1

Then “Enter Password:” text is written to the console’s standard output handle using `WriteFile()`, and user input is read from the console standard input handle via `ReadFile()`. The input password is stored in the stack variable that IDA labels as `Buffer` and can be renamed to “input_password”.

```
push    0                ; lpOverlapped
lea     eax, [ebp+NumberOfBytesWritten]
push    eax              ; lpNumberOfBytesWritten
push    12h              ; nNumberOfBytesToWrite
push    offset aEnterPassword ; "Enter password:\r\n"
mov     ecx, [ebp+stdout]
push    ecx              ; hFile
call    ds:WriteFile
push    0                ; lpOverlapped
lea     edx, [ebp+NumberOfBytesWritten]
push    edx              ; lpNumberOfBytesRead
push    80h              ; nNumberOfBytesToRead
lea     eax, [ebp+input_password]
push    eax              ; lpBuffer
mov     ecx, [ebp+stdin]
push    ecx              ; hFile
call    ds:ReadFile
```

Figure 6: Console output and input in challenge1

This password input is then used as a parameter to an unknown function `sub_401260`, along with the password length. The result from this function is compared with the interesting string in the variable we named “`important_string`”, to determine whether the password is correct. This comparison happens in the `_strcmp()` function (at address `0x402c30`). If the return value (in register `eax`) from `_strcmp` is zero, this means the strings were equal, and we will follow the path to writing “Correct!” to the console.

Function `sub_401260` is therefore responsible for massaging the password input into a form that can be compared with “`important_string`” and is the key to solving this challenge. We can name this function “`modify_password`”, and its return value “`modified_password`.”

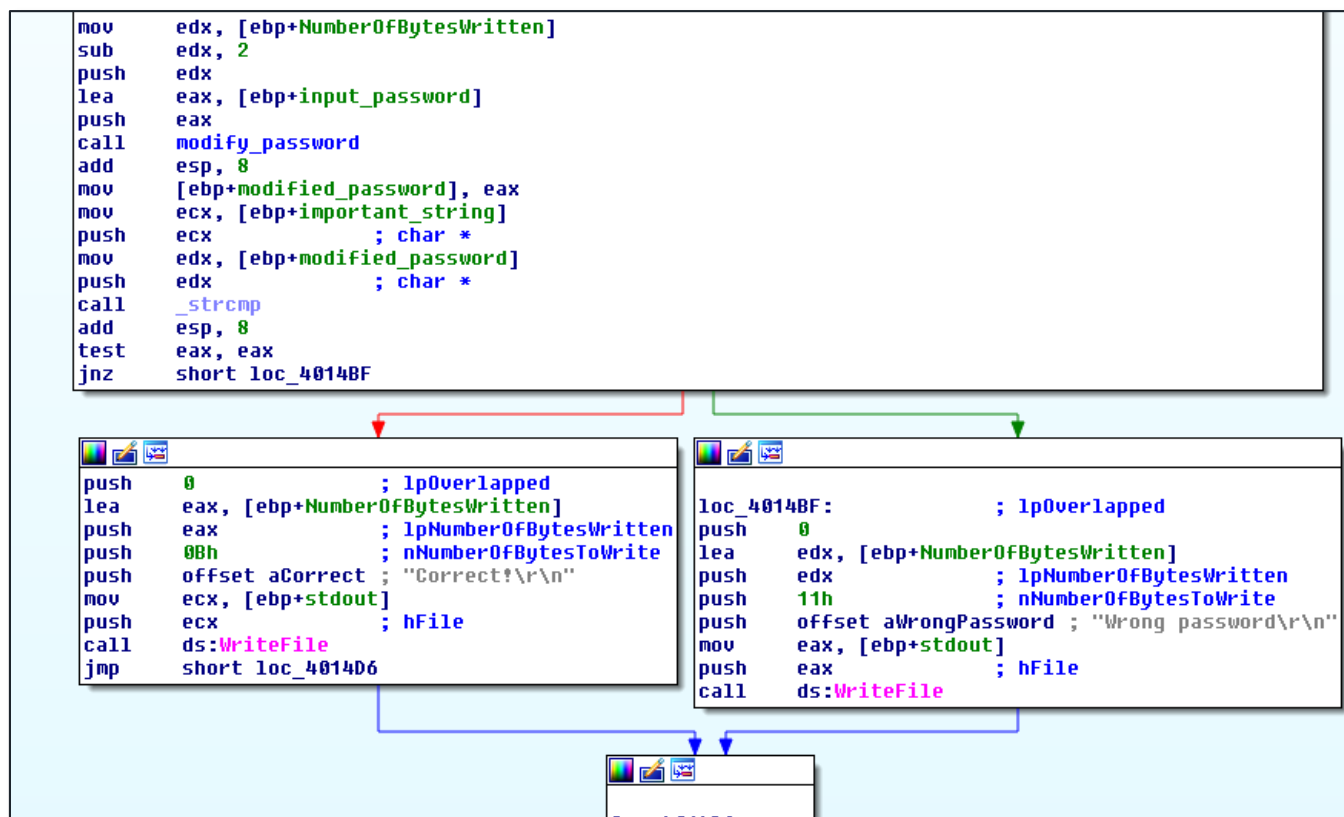


Figure 7: Modify password and compare with important string

The `modify_password` function will initially appear complicated. However, there are a couple of clues we can look for to figure out a standard function is being used here. First, the reference to constant `0x3d`, which is the ASCII character code for `'=`.

Second, references to a 64 character array at `0x413000` (which points to the second interesting string that was identified earlier with the strings tool).

```

mov    ecx, [ebp+var_18]
sub    ecx, 1
sub    ecx, [ebp+var_14]
mov    edx, [ebp+var_C]
mov    byte ptr [edx+ecx-1], 3Dh
jmp    short loc_4013DB

```

Figure 8: Reference to 0x3d (=) padding character in the modify password function

```

mov    ecx, 0Fh
mov    edx, [ebp+var_C]
add    edi, [ebp+var_8]
mov    a, byte_413000[ecx]
mov    [edx], a
mov    ecx, [ebp+var_8]
add    ecx, 1
mov    [ebp+var_8], ecx
mov    edx, [ebp+var_10]
shr    edx, 0Ch
and    edx, 3Fh
mov    eax, [ebp+var_C]
add    eax, [ebp+var_8]
mov    c, byte_413000[edx]
mov    [eax], c
mov    edx, [ebp+var_8]
add    edx, 1
mov    [ebp+var_8], edx
mov    eax, [ebp+var_10]
shr    eax, 6
and    eax, 3Fh
mov    ecx, [ebp+var_C]
add    ecx, [ebp+var_8]
mov    d, byte_413000[eax]
mov    [eax], d
mov    eax, [ebp+var_8]
add    eax, 1
mov    [ebp+var_8], eax
mov    ecx, [ebp+var_10]
shr    ecx, 0
and    ecx, 3Fh
mov    edx, [ebp+var_C]
add    edi, [ebp+var_8]
mov    a, byte_413000[ecx]
mov    [edx], a
mov    ecx, [ebp+var_8]

```

Figure 9: References to 64 character index string in the modify password function

These observations would suggest that this is the Base64 algorithm, which typically uses “=” as a padding character and uses a 64 character array index.

An experienced malware analyst may also have noticed that the first interesting string, `x2dtJEOmyjacxDemx2eczT5cVS9fVUGvWTuZWjueXjRqy24rV29q`, is made up of a pattern of both lowercase and uppercase letters as well as numerals and guessed that it was a Base64 encoded string.

Base64 decoding can be implemented fairly easily in python with the following script:

```
import base64
encoded_string = "x2dtJEOmyjacxDemx2eczT5cVS9fVUGvWTuZWjueXjRqy24rV29q"

print base64.b64decode(encoded_string)
```

Figure 10: Regular Base64 Script

However the output from this is unfortunately nonsensical. The clue to solving this issue is in the second interesting string:

`ZYXABCDEFGH IJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/`

This is the 64 character byte array at 0x413000. A normal Base64 algorithm uses the following indexing string in the encoding process:

`ABCDEFGHI JKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/`

This challenge has modified that indexing string by moving a few letters around, effectively making it into a substitution cipher, so we will have to account for this in the python script by adjusting our input string to compensate.

```
import base64, string
encoded_string = "x2dtJEOmyjacxDemx2eczT5cVS9fVUGvWTuZWjueXjRqy24rV29q"

translated_encoded_string = encoded_string.translate(
string.maketrans("ZYXABCDEFGHJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/",
"ABCDEFGHIJKLMNPOQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"))
```

Figure 11: Base64 Script with modified alphabet

This second script produces the correct output: sh00ting_phish_in_a_barrel@flare-on.com

We can then test this on the challenge to verify its accuracy:



Figure 12: Screenshot of challenge1 correct password